Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 10 (2012) 30 – 37

Procedia Computer Science

The 3rd International Conference on Ambient Systems, Networks and Technologies (ANT)

S-CLAIM: An Agent-based Programming Language for AmI, A Smart-Room Case Study[☆]

Valentina Baljak[b], Marius Tudor Benea[a], Amal El Fallah Seghrouchni[a], Cédric Herpson[a,*], Shinichi Honiden[b], Thi Thuy Nga Nguyen[a], Andrei Olaru[c], Ryo Shimizu[b], Kenji Tei[b], Susumu Toriumi[b]

[a]*LIP6 - University Pierre and Marie Curie, France*
[b]*NII - University of Tokyo, Japan*
[c]*University Politehnica of Bucharest, Romania*

Abstract

This paper introduces a declarative agent-oriented language for Ambient Intelligence – S-CLAIM – that allows programming reactive or cognitive mobile agents in a simple, easy-to-use manner while meeting AmI requirements. Based on a hierarchical representation of the agents, the language offers a natural solution to achieve context-sensitivity. S-CLAIM is light-weight and, being transparently underpinned by the JADE framework, allows deployment on mobile devices and easy interoperation with other components by means of web services. The usefulness of the proposed language for AmI is illustrated through a scenario and a demo featuring an AmI application in a Smart Room.

Keywords: Ambient Intelligence, Programming Languages, Multi-Agent Systems

1. Introduction

Ambient Intelligence (AmI) applications are characterized by the intrinsic distribution of their architecture, the dynamic of their topologies and the frequent changes in their execution context. At a behavioral level, the need for context-sensitivity is a key element, allowing AmI applications to adapt to the various situations and users they may encounter. It is therefore natural that Multi-Agent Systems and the Agent-Oriented Paradigm (AOP) have emerged as a well suited approach for the implementation of AmI applications [1, 2].

Thus, different architectures have been used for organizing agents in AmI systems. However, existing agent-oriented languages rely on various underlying frameworks for the effective implementation and execution of the agents. Most of these platforms do not natively offer a way to represent and manipulate the behaviors of the agents as web services [3, 4]. That restricts the use of agent-oriented approaches in the

[☆]This work has been realised under a MoU between UPMC and NII.
[*]Corresponding author.

context of AmI. Indeed, AmI applications require to support and connect different devices, from computers to sensors, which is usually done by representing them as web-services.

In order to allow the representation of cognitive skills such as beliefs, goals and knowledge (like the current execution context), while meeting the requirements of mobile computation and execution in smart environments, we present in this paper a high-level declarative AOP language: S-CLAIM.

Using the hierarchical representation of the agents inherited from CLAIM combined with new features, S-CLAIM goes beyond the limitations of current languages. Thus, S-CLAIM allows programmers to use the agent-oriented paradigm during the whole process of designing and implementing an AmI application, as S-CLAIM specifies only agent-related components and operations, leaving algorithmic processes aside, and also due to the fact that S-CLAIM agents are – transparently to the programmer – interpreted and executed on top of the JADE [5] framework, which handles communication, mobility, and agent management. This latter choice, combined with the expressiveness of S-CLAIM, offers lightweight agents, cross-platform deployment and mobile device compatibility, currently implemented for the Android platform.

This paper is structured as follows. Section 2 presents some related works. Section 3 introduces the general structure of S-CLAIM and details the syntax and the semantics of the primitives necessary in a smart environment context. We then illustrate the usefulness of S-CLAIM through the *Smart Room* case study in Section 4, going from scenario description to its actual execution. Finally, we draw the conclusions and we present the current limitations of our work and some perspectives to extend it, in Section 5.

2. Related Work

As with any programming paradigm, several languages have been proposed for the implementation of agents, ranging from purely imperative to purely declarative, including various hybrid approaches [6]. Depending on the language considered, the modelling of the agents, behaviors, and knowledge (such as the current context) varies.

On one hand, the agent-oriented programming (AOP) languages, such as AgentSpeak [7], or 3APL allow to represent the mental state of the agents, but do not support the agents' mobility. Moreover, in the manner of AgentSpeak, most of the existing languages represent the current context of the agents as a set of facts [8]. If this representation is well suited for most of commonly used agent applications, this is not the case in the context of AmI. Indeed, this representation does not allow to represent the dependency relationships between the different elements that form the context, or between agents when they are perceived as an integral part of the environment.

On the other hand, concurrent languages such as the ambient calculus [9] have been proposed to formalize concurrent and mobile processes in distributed environments. They have a well defined operational semantics, but it is impossible to represent intelligent agents using them.

The CLAIM language [10] combines in a unified framework the main advantages of AOP languages, for representing the mental state of the agents, with those of the concurrent languages for representing the concurrence and mobility of the agents. However, some of the aspects of CLAIM restrict its usability in real-life AmI situations. Indeed, the weight of the CLAIM agents and of the application layer necessary to deploy them does not allow to use it on networks composed of devices with low memory or low computational capabilities. Moreover, as the behaviors of the CLAIM agents were not conceived with a web services perspective in mind, there is currently no possibility to deploy or use them on a network combining heterogeneous devices and platforms.

S-CLAIM (Smart Computational Language for Autonomous, Intelligent and Mobile Agents) is the spiritual descendant of the CLAIM language and tries to go beyond these limitations.

3. S-CLAIM

In order to fulfill the previously introduced requirements of a language for AmI, S-CLAIM uses an evolution of the operational semantics of CLAIM to describe the behavior of the multi-agent system. S-CLAIM allows various representations of the knowledge base, as long as they can be addressed by relations or association patterns. This flexibility facilitates the design and implementation of proactive agents and goal-oriented behaviors. To allow cross-platform deployment and mobile device compatibility, S-CLAIM agents are – transparently to the S-CLAIM programmer – interpreted and executed on top of the JADE [5] Framework. To allow the users to keep themselves away from the algorithmic complexity, S-CLAIM agents are programmed using a high-level language, based on a Lisp-like syntax. To facilitate cross platform mobility, the knowledge and the description of the scenarios are stored in XML files. Moreover, the hierarchical organization of the agents makes them more flexible and adaptive. Thus, each agent belongs to a hierarchy of agents. Its parent and children depend of its current role and execution context. Every agent can move within its hierarchy or to a remote one depending on the evolution of the services it provides and on the achievement of its goals. As illustrated in Figure 1, when an agent moves, it moves as a whole, with all its components (intelligent elements, running processes and sub-agents) maintaining, thus, the agents' dependency relationships.

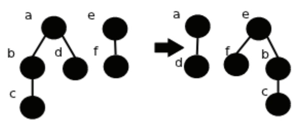


Fig. 1. Migration of agent *b* and his subtree.

Some of the most important parts of an agent are its behaviors. They define what an agent can do in certain situations. There are three types of behaviors in S-CLAIM. *Initial* behaviors are run immediately after the agent was created, handling all needed initializations. *Reactive* behaviors are triggered by the reception of messages that respect certain templates. *Proactive* behaviors are cognitive and goal-oriented. In order to define these behaviors, the S-CLAIM language uses the primitives shown in Table 1. Their syntax and semantics will be detailed in the following sections.

Table 1. The primitives of S-CLAIM.

Messaging primitives	
send	sending message
receive	receiving message
Mobility primitives	
in	migrating to a hierarchy
out	exiting its parent hierarchy
Knowledge management primitives	
addK	adding a new knowledge record
removeK	removing a knowledge record
readK	accessing a knowledge record
forAllK	extracting knowledge from kb

Agent management primitives	
open	ordering agent to dissolve
acid	dissolving itself
new	creating a new agent
Control primitives	
condition	triggering condition of a behavior
if	condition control inside the behavior
wait	waiting for a certain amount of time
Goal-oriented primitives	
aGoal	state wanted to be achieved
pGoal	activities to be performed
mGoal	state wanted to be maintained

3.1. Syntax

S-CLAIM was designed to have a very simple syntax. We chose, accordingly, to adapt to the purposes of our language a fully parenthesized Lisp-like syntax, that is presented in this section.

Each construct contains two limiting parenthesis, an identifier (either a keyword or a function name) and a list of arguments. The production rule of an agent class specification has the following form:

```
agent_class_specification -> '(' "agent" class_name agent_args_list behaviors ')'
```

We have, in the above production rule, the keyword *agent*, followed by the list of arguments (the first one being the name of the agent class) and a list of behaviors.

The variables of the language are strings that start with the character ?, followed by a number of letters or digits. The constants are strings of any character except ?, *(*, *)*, or separators (spaces, tabs, new lines). The syntax for comments is similar to the syntax in Java or C++.

Below, an example of a reactive behavior that registers a room and its associated agent is shown.

```
(reactive registerRoom
    (receive managesRoom ?agentName ?roomName)
    (addK (struct knowledge roomAgent ?roomName ?agentName))
)
```

This behavior, *registerRoom*, contains two statements. The first one specifies that the behavior should be triggered when receiving a message composed by the constant *managesRoom* followed by two variables. When triggered, the knowledge record *(struct knowledge roomAgent ?roomName ?agentName)* is stored in the knowledge base of the agent, using the *addK* statement.

The *receive* and *condition* constructs are constrained to appear only at the beginning of the behavior. We imposed this restriction for code readability reasons. It is the only restriction of this type.

For most of the constructs of S-CLAIM, the list of arguments can contain variables, constants, structures or *function calls*. The syntax of the function calls is similar to the one of the other constructs, with the difference that the identifier of the statement is not a keyword. Functions contain algorithmic processings and are implemented in other, more adequate, languages than S-CLAIM (Java is currently supported). Functions calls handle only processings that does imply any components of the agent (communication, knowledge, etc).

Structures may represent knowledge, messages, or any other group of values. The fields of the structures could be composed of variables, constants or other structures. An example showing the syntax of a structure is presented below:

```
(struct knowledge userAgent ?userName ?agentName)
```

This example represents a knowledge record that specifies the association of type *userAgent* between the user's name and the name of the agent that assists him.

In general, the arguments are variables, constants, structures or function calls. However, some primitives, like *if*, *condition* and *forAllK*, could also take other S-CLAIM constructs as arguments.

3.2. Semantics

S-CLAIM uses a semantic inspired by ambient calculus and by π-calculus to cover the important aspects of an intelligent agent such as reasoning, asynchronous communication, concurrence and mobility. In the following, all the primitives used in S-CLAIM (see Table 1) are briefly presented.

Messaging primitives. The ***send*** primitive takes at least 2 parameters: the receiver(s) of the message and its content, represented by a message structure. The interoperability with web services has been integrated seamlessly in the language and platform, using the existing primitives. All S-CLAIM behaviors are, thus, exposed as web services, and all S-CLAIM agents are able to invoke web services using a modified *send* primitive, that uses the address of the destination, and a receiving structure for the response (if expected):

```
(send ?Ag (struct message echo) http://localhost/wsig/ws/ (struct message ?back))
```

The ***receive*** primitive will check if any received message matches the message pattern present in the definition. If they match, and any subsequent *condition* constructs are satisfied, the agent will activate the behavior and will bind the variables in the *receive* primitive to their values, received by the message. In S-CLAIM the communication between the agents is asynchronous.

Mobility and agent management primitives. The primitive ***in*** moves an agent to a new ambient, i.e. a new hierarchy. The agent will become a child of the agent given as argument. When an agent moves, all its children in the hierarchy are notified and – depending on their dependency relationships – instructed to follow and to move to the new hierarchy. The primitive ***out*** is used to quit the current hierarchy. The agent (and its subtree) will no longer be the child of its old parent, but of its parent's parent. The ***open*** primitive can be used by an agent to absorb one of its children, and the agent will recover all the components (knowledge, children, behaviors) of this child. The ***acid*** primitive is used to dissolve the agent itself; all its components will become the components of its parent. Finally, the ***new*** primitive creates a new child agent.

Knowledge management primitives. The ***addK*** primitive is used to add knowledge to the knowledge base while the ***removeK*** primitive does the opposite. The ***readK*** primitive searches the knowledge base for the existence of knowledge entries that match a given pattern and extracts the first matching knowledge record. Unlike *readK*, the ***forAllK*** primitive is used to extract *all* the knowledge entries in the knowledge base that match a given pattern. See how these primitives are used in the example code in Figure 2.

Control primitives. Always placed at the beginning of the behaviors, the ***condition*** primitives can verify if it is possible to execute a triggered behavior. The ***if-else*** statement executes, based on a condition, the block of statements corrensponding to the selected branch. Both *condition* and *if-else* are used in conjunction with function calls that return Boolean values, or with the *readK* primitive, that returns *true* if any knowledge entry matching the given pattern has been found. The ***wait*** primitive is used to suspend the current behavior for a certain amount of time.

Goal-oriented primitives. Associated with the manipulation of the mental states of the agents, they were implemented in S-CLAIM based on the three goal types proposed by Braubach *et al.*: *Perform*, *Achieve* and *Maintain* [11]. The life cycle of the goals comprises three main states, *New*, *Adopted* (with the substates *Option*, *Active* and *Suspended*) and *Finished* (inspired by Dastani *et al.* [12]). Additionally, a model to express the priorities of the goals and a way to represent, hierarchically, the graph structure of the goal base were proposed. All these were packed inside the S-CLAIM's proactive behavior type.

3.3. Example of an S-CLAIM agent definition

Consider an example in which, in a certain scenario, the user has to display some opinions, from his/her PDA, on a screen. The agent that assists the user, running on the PDA, has the class PDAagent. In S-CLAIM, a program is composed of a scenario (described in an XML file) and of some agent class definition files. Figure 2 presents the description file of the agent class for this example.

The initial behavior *register* informs the agent's parent, at creation, that it assists the user denoted by the variable *?userName*. The parent's name, as well as the name of the user to assist, are received as arguments, after they were read from the scenario's XML file (not presented here).

The reactive behavior *assignScreen* is triggered when receiving a message of type *screenAssigned*, containing a variable. When triggered, it is verified if the agent already has a screen assigned. If *true*, it is verified if the old screen and the new one are different. If positive, the agent managing the old screen is informed to remove the rights of the user to display information on the screen. The old screen is also removed from the knowledge base of the agent. Then it moves to the sub-hierarchy of the agent managing the new screen. It also stores the information about the new screen in the knowledge base. All the opinions, together with their types, are sent to the agent managing the new screen and they will be displayed. If no screen was previously assigned, a sequence of code identical with the one from the lines 15-20 is executed.

Note that in S-CLAIM, once variables are bound inside a context (agent, behavior, block), they keep their value until the end of their context. Any further test on the variable will consider it as bound, therefore as a restriction to a pattern.

The *isDifferent* function, that is used at line 11, is a Java function from an external library.

4. Case study: Smart Room

In this section, we illustrate the use of S-CLAIM for AmI, from scenario description to actual execution.

```
 1  (agent PDAagent ?userName ?parent
 2    (behavior
 3      (initial register
 4        (send ?parent (struct message assistsUser this ?userName))
 5      )
 6      ....
 7      (reactive assignScreen
 8        (receive screenAssigned ?screenAgentName)
 9        (if (readK (struct knowledge useScreen ?oldscreenAgentName))
10        then
11          (if (isDifferent ?oldscreenAgentName ?screenAgentName)
12          then
13            (send ?oldscreenAgentName (struct message removeUser ?userName))
14            (removeK (struct knowledge useScreen ?oldScreenAgentName))
15            (in ?screenAgentName)
16            (addK (struct knowledge useScreen ?screenAgentName))
17            (readK (struct knowledge opinionType ?type))
18            (forAllK (struct knowledge opinion ?opinion)
19              (send ?screenAgentName (struct message opinionList ?type ?userName ?opinion))
20            )
21          )
22        else
23          *the content from the lines 15-20, repeated*
24        )
25      )
26    )
27  )
```

Fig. 2. Example of an agent definition

4.1. Scenario description

The following scenario aims to highlight the two intrinsic characteristics of an AmI system: context-awareness and anticipation [13, 14].

Scenario. Alice is a student at the university. Today, the Multi-Agent Systems (MAS) course is held in a room other than usual. All the students of this class are notified automatically via their smartphones about this change and receive an indication on how to get to the new room. Alice is the first one who arrives. While she enters, the lights are automatically turned on and the main screen shows a welcome message. When it is time to start the course, observing that the professor and all the students are in the room, the lights dim and the main screen shows the first slide of the presentation. When the professor indicates that the presentation has finished, the lights turn on again to start the second section of the course: brainstorming. The class is divided into several groups. Each group has a large smart screen to display their opinions. Students write their opinions on their smartphone or laptop. The opinions appear right away on the screen associated to the group, so that the others could see them. When Alice moves to another group to discuss, her opinions are automatically displayed on the screen of the new group, and removed from the other one.

4.2. Modeling the scenario

The scenario is modeled according to the AOP paradigm. Thus, each entity has an associated agent, that represents it. A graphical representation of the agentification of the scenario can be seen in Figure 3.

According to the S-CLAIM approach, all the agents are part of one hierarchy (that can be read from left to right in Figure 3, starting with the highest level). Thus, the root agent in the hierarchy is *University*. It manages high level information about the university, like the campuses and their locations. Further, the agents associated to the campuses store information about the rooms, whose associated agents keep track of all the devices inside them and the managing agents, and so on. All the information needed about the university is stored in this hierarchy of agents, so, if one agent needs to know something that is not stored in its knowledge base, it will ask his parent for that information. This process is recursive.

As specified in the legend, the agents belong to four different classes of roles. In Figure 3, each class has a different color. The edges are labeled with the relations between the involved agents and belong, too, to

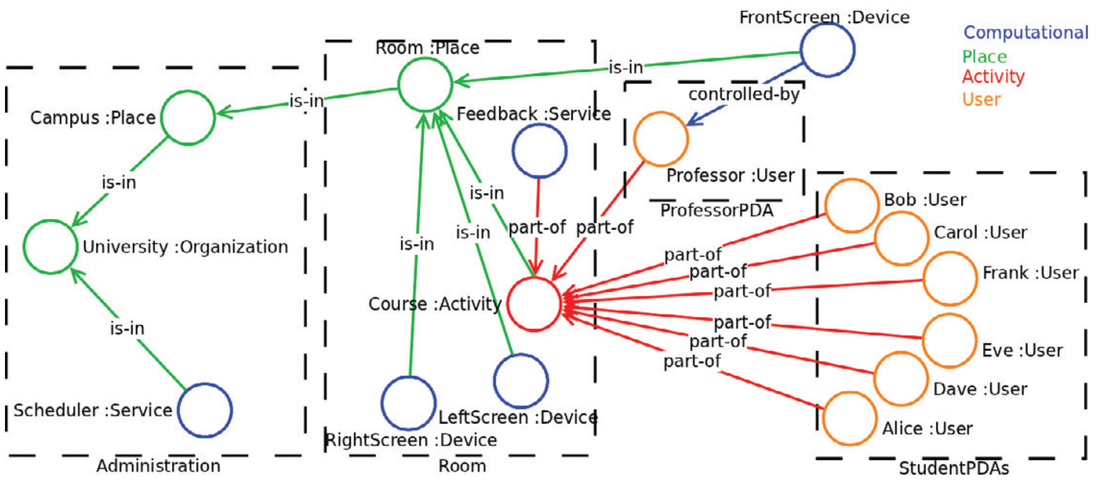


Fig. 3. Scenario agentification example

one such class, according to their color. For example, the *Feedback* agent is *part-of* the activity represented by the *Course* agent. Such a graph describes the structure of the system in only one particular moment.

The rectangles indicate the computing devices on which the agents included in them run. Note that the FrontScreen agent is only presented out of the *Room* for readability reason.

4.3. Experiment

S-CLAIM is part of the AoDai framework [13]. In order to study S-CLAIM's efficiency for real-life AmI applications, we have deployed the system in the *Smart Room* located at National Institute of Informatics in Tokyo. The Smart Room has ten projectors, two LCDs, 2 speakers, and a dozens of lights. These devices are connected to a network, and can be controlled through a RESTful web service interface. In addition, a wireless sensor network consisting of a dozen of Oracle's SunSPOT sensor nodes are installed in the Smart Room. The SunSPOT nodes contain radio sensors and can detect radio signal propagated from a beaconing device that each student has. By using RSSI-based localization, the systems can detect the presence and location of students. These devices and the wireless sensor network are managed by S-CLAIM agents which communicate with the components using the WSIG and WSDC web services add-ons for Jade.

Figure 4 (a) shows a picture of the *Smart Room* of the National Institute of Informatics, and Figure 4 (b) shows a screenshot of the Android device of Bob, running its assistant agent. The screenshot was taken right after the creation of the agent – Bob's agent has just registered with its parent, *CourseCSAgent*. A more detailed example can be seen through a video available on our website[1].

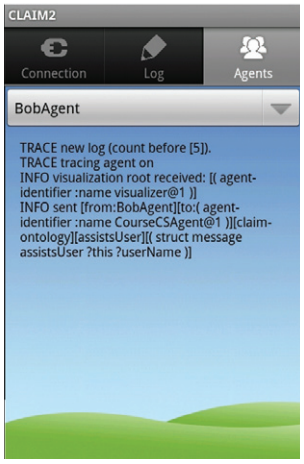
5. Conclusion and perspective

In this paper we have presented the S-CLAIM agent-oriented programming language, that allows a designer of AmI applications to program agents in a simple and intuitive manner. Whithout dealing with low-level considerations and using only a small number of primitives focused on agent-specific features, the programmer can focus on communication, mobility, knowledge and agent management.

The platform that underpins the S-CLAIM agents handles agent operations, function calls, web service integration and deployment on mobile devices. The effectiveness and the usefulness of the language and of the platform in an AmI context have been already proven - both in term of number of lines of code and development time - in a first application implemented using the Smart Room.

[1] `http://webia.lip6.fr/~aodai/videos/aodai.wmv`

(a)



```
TRACE new log (count before [5]).
TRACE tracing agent on
INFO visualization root received: [( agent-
identifier :name visualizer@1 )]
INFO sent [from:BobAgent][to:( agent-
identifier :name CourseCSAgent@1 )][claim-
ontology][assistsUser][( struct message
assistsUser ?this ?userName )]
```

(b)

Fig. 4. (a) Smart room in National Institute of Informatics. (b) Screen capture of S-CLAIM for Android.

However, a number of points have to be studied in-depth. First, the language and the platform need to be used for the implementation of more complex scenarios and AmI applications in order to fully test and validate their usability in real situations. Also, the number of primitives of S-CLAIM has been deliberately limited to the necessary ones. In order to improve the flexibility of S-CLAIM, an important feature will be to allow the AmI application programmers to easily extend the language by defining their own primitives.

References

[1] F. Sadri, Ambient intelligence: A survey, ACM Comput. Surv. 43 (4) (2011) 36:1–36:66.

[2] C. Ramos, J. C. Augusto, D. Shapiro, Ambient intelligence - the next step for artificial intelligence, IEEE Intelligent Systems 23 (2) (2008) 15–18.

[3] M. Lyell, L. Rosen, M. Casagni-Simkins, D. Norris, On software agents and web services: Usage and design concepts and issues, in: Proc. of the 1st International Workshop on Web Services and Agent Based Engineering, Sydney, Australia, 2003.

[4] M. Shafiq, Y. Ding, D. Fensel, Bridging multi agent systems and web services: towards interoperability between software agents and semantic web services, in: Enterprise Distributed Object Computing Conference, 2006. EDOC'06. 10th IEEE International, IEEE, 2006, pp. 85–96.

[5] F. Bellifemine, A. Poggi, G. Rimassa, Developing multi-agent systems with JADE, Intelligent Agents VII Agent Theories Architectures and Languages (2001) 42–47.

[6] R. Bordini, L. Braubach, M. Dastani, A. El FSeghrouchni, J. Gomez-Sanz, J. Leite, G. O Hare, A. Pokahr, A. Ricci, A survey of programming languages and platforms for multi-agent systems, INFORMATICA-LJUBLJANA- 30 (1) (2006) 33.

[7] A. Rao, Agentspeak (l): Bdi agents speak out in a logical computable language, Agents Breaking Away (1996) 42–55.

[8] I. Ayala, M. Pinilla, L. Fuentes, Modeling context-awareness in agents for ambient intelligence: an aspect-oriented approach, Progress in Artificial Intelligence (2011) 29–43.

[9] L. Cardelli, A. D. Gordon, Mobile ambients, Theor. Comput. Sci. 240 (1) (2000) 177–213.

[10] A. Suna, A. El Fallah Seghrouchni, Programming mobile intelligent agents: An operational semantics, Web Intelligence and Agent Systems 5 (1) (2004) 47–67.

[11] L. Braubach, A. Pokahr, D. Moldt, W. Lamersdorf, Goal representation for bdi agent systems, in: R. Bordini, et al. (Eds.), Programming Multi-Agent Systems, Vol. 3346 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2005, pp. 44–65.

[12] M. Dastani, M. B. van Riemsdijk, J.-J. C. Meyer, Goal types in agent programming, in: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems, AAMAS '06, ACM, New York, NY, USA, 2006, pp. 1285–1287.

[13] A. El Fallah Seghrouchni, A. Olaru, T. T. N. Nguyen, D. Salomone, Ao Dai: Agent oriented design for ambient intelligence, in: N. Desai, A. Liu, M. Winikoff (Eds.), Principles and Practice of Multi-Agent Systems, 13th International Conference, PRIMA 2010, Kolkata, India, November 12-15, 2010, Revised Selected Papers, Vol. 7057 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2011, pp. 259–269.

[14] A. Olaru, A. El Fallah Seghrouchni, A. M. Florea, Ambient intelligence: From scenario analysis towards a bottom-up design, in: M. Essaaidi, M. Malgeri, C. Badica (Eds.), Intelligent Distributed Computing IV, Proceedings of the 4th International Symposium on Intelligent Distributed Computing - IDC 2010, Tangier, Morocco, September 16-18 2010, Vol. 315 of Studies in Computational Intelligence, Springer Berlin / Heidelberg, 2010, pp. 165–170.